Управление памятью

Статическое управление памятью

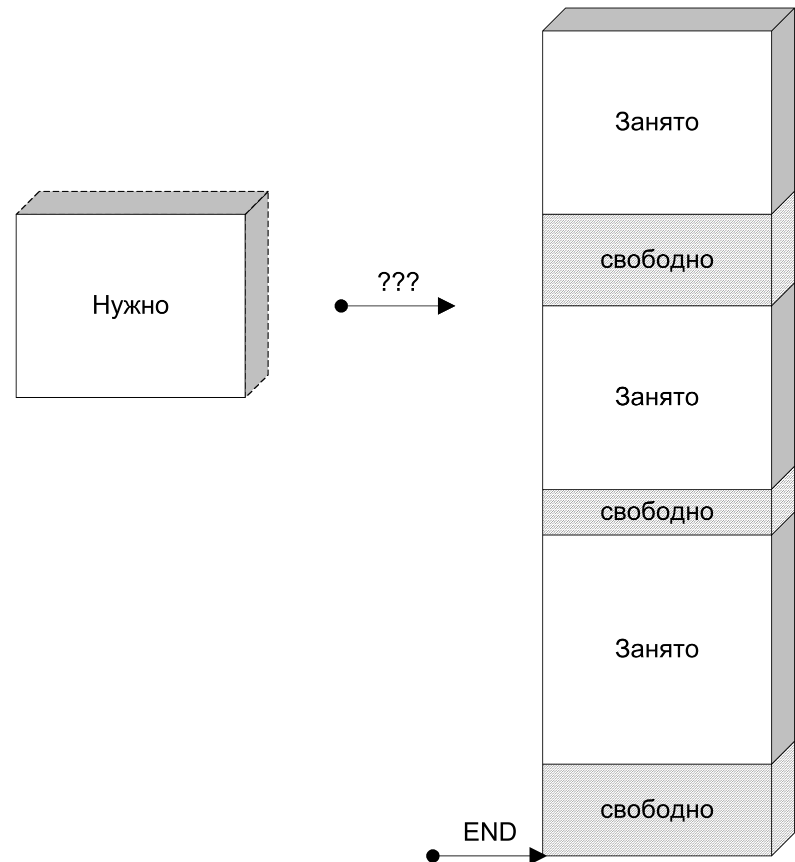
- Требуемое количество памяти известно в момент сборки программы
- Достаточно проверить, есть ли требуемая память в момент загрузки
- Достаточно помнить границу между занятой и свободной памятью

Динамическое управление памятью

- Память может запрашиваться и освобождаться во время исполнения программы
- Необходимо поддерживать список областей свободной памяти (*пул, куча*)
- Пример вырожденного пула – стековые кадры функций и методов в C/C++
- (Дополнительно) стековое (LIFO) управление памятью при загрузке программ в DOS

Внешняя фрагментация

- Без возможности перемещать блоки по памяти, проблема неразрешима.
- Оценка Кнута: при случайном размере блоков и случайном порядке выделения и освобождения, при установившемся размере пула 1/3 его объема – свободные блоки



Внутренняя фрагментация

- Выделять память блоками, размер которых кратен килобайту
- Если программе нужно 646 байт, 378 байт теряется
- Оценка величины потерь – половина килобайта на каждый блок (точная оценка требует интегрирования функции распределения размеров блоков)
- Ограничения на размер выделяемых блоков могут ограничить внешнюю фрагментацию, но при этом создают внутреннюю фрагментацию

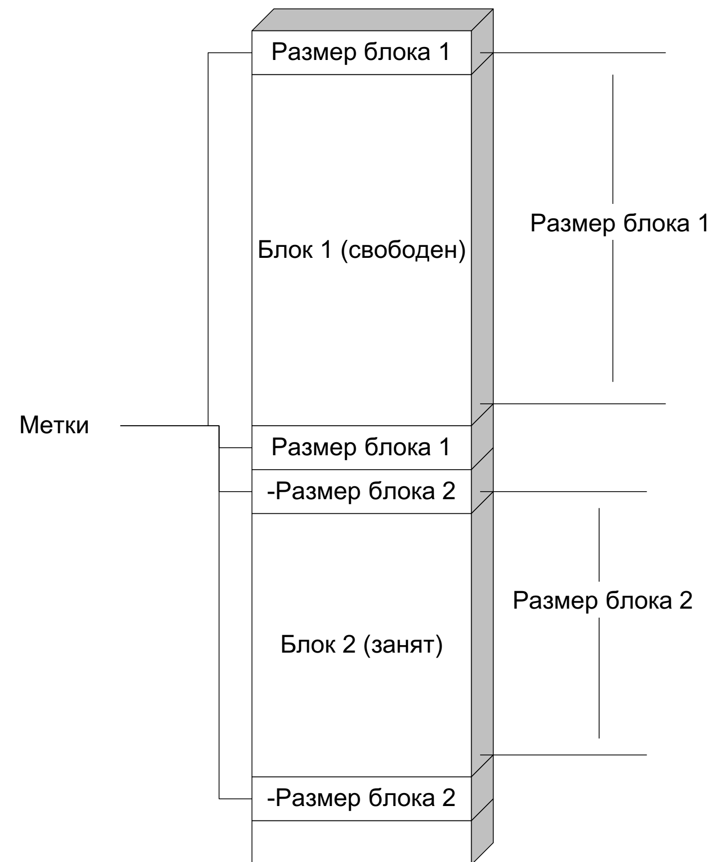
Внутренняя фрагментация

Стратегии поиска свободных блоков

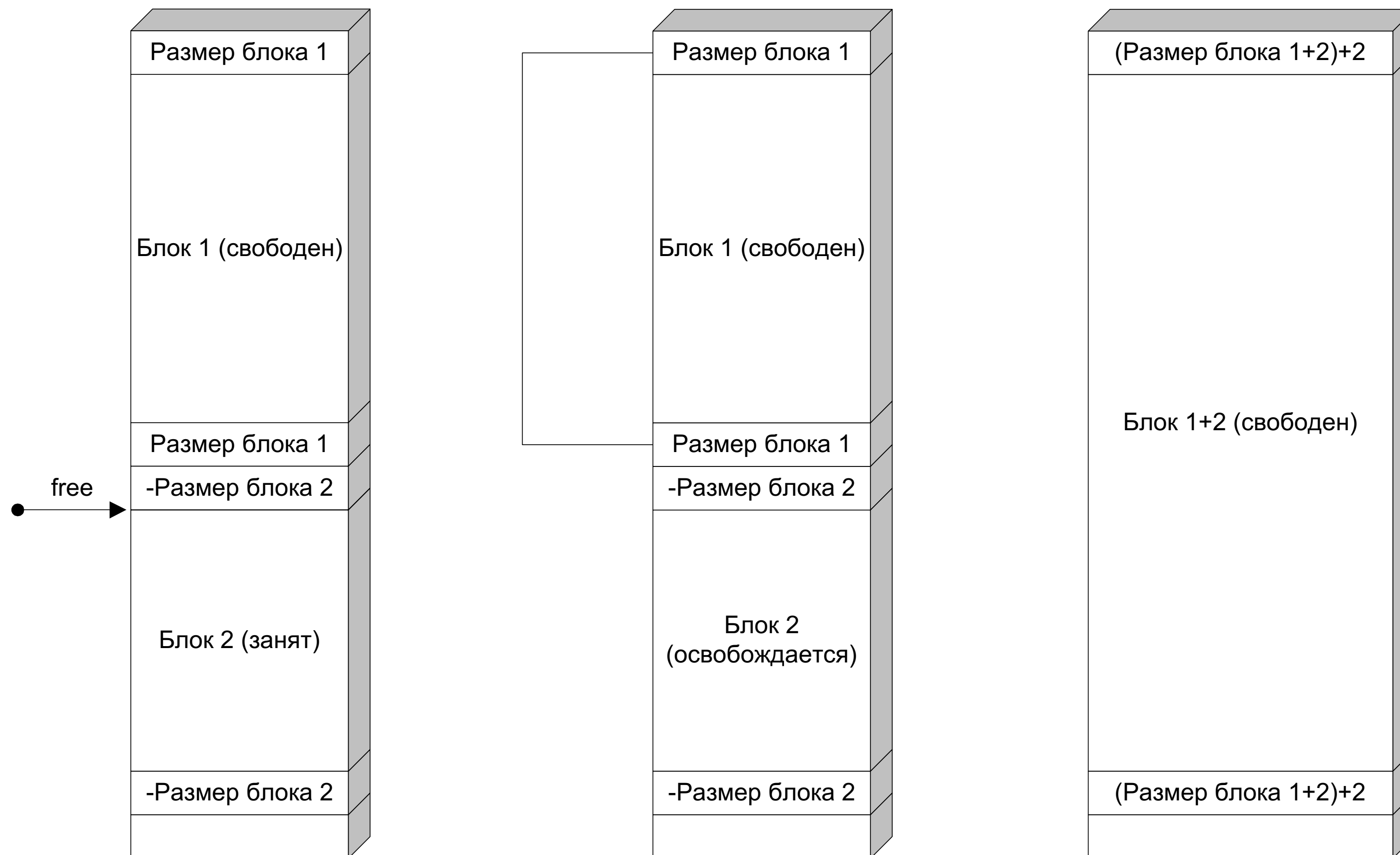
- Наиболее подходящий (Best Fit)
  - Требует просмотра всего пула либо сортировки
  - Увеличивает фрагментацию (минимизирует средний размер отбрасываемого «хвоста»)
- Первый подходящий (First Fit)
  - Лишен недостатков Best Fit
  - Используется чаще всего
- Наименее подходящий (Worst fit)
  - Требует сортировки блоков по размеру
  - Используется в файловых системах (напр. HPFS)

Алгоритм парных меток

- Блок окружается метками (дескрипторами), указывающими состояние блока (занят/свободен) и его размер
- Можно рассматривать, как «естественно» сортированный по адресам двусвязный список



Объединение блоков в алгоритме парных меток



Реализации malloc(3C) в Solaris (OpenIndiana)

- malloc(3C) в libc (используется по умолчанию)
- bsdmalloc, malloc(3MALLOC) (libmalloc), mtmalloc, libumem
- См. malloc(3C), секция USAGE и umem_alloc(3MALLOC), секция NOTES, чем именно они отличаются
- watchmalloc (отладочная версия)
- lmalloc (внутренний аллокатор для нужд libc).

Зачем столько?

- Некоторые старые приложения оптимизированы по производительности под поведение определенных реализаций malloc
  - Или просто полагаются на баги в этих реализациях
- Старые реализации malloc плохо себя ведут в многопоточных программах
- Масштабируемые версии (mtmalloc, libumem) сложнее и имеют свои приколы

Что еще бывает?

- Старый malloc из glibc
(описан в моей книге, выкошен где-то после 2010 года из-за проблем с многопоточностью)
- ptmalloc, используется в современных версиях GNU libC
https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html
- Jemalloc, используется в BSD, MacOS и Firefox

Реализация malloc(3C) в OpenIndiana libc

- https://github.com/illumos/illumos-gate/blob/master/usr/src/lib/libc/port/gen/malloc.c
- https://github.com/illumos/illumos-gate/blob/master/usr/src/lib/libc/port/gen/mallint.h
- Почти чистый алгоритм парных меток
- Но использует балансированное дерево вместо списка
- На самом деле, best fit
  (хотя в комментариях написано что first fit)

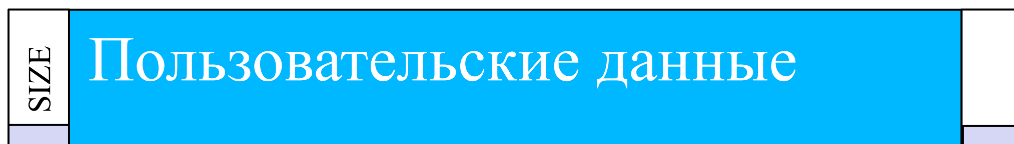
Дескриптор блока

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
   size_t w_i; /* an unsigned int */
   struct _t_ *w_p; /* a pointer */
   char w_a[ALIGN]; /* to force size */
} WORD;

/* structure of a node in the free tree */
typedef struct _t_ {
   WORD t_s; /* size of this element */
   WORD t_p; /* parent node */
   WORD t_l; /* left child */
   WORD t_r; /* right child */
   WORD t_n; /* next in link list */
   WORD t_d; /* dummy to reserve space for self-pointer */
} TREE;

/* usable # of bytes in the block */
#define SIZE(b) (((b)->t_s).w_i)
```

Дескриптор блока

```
/* functions to get information on a block */
#define  DATA(b)        ((char *)(((uintptr_t)(b)) + WORDSIZE))
#define  BLOCK(d)       ((TREE *)(((uintptr_t)(d)) - WORDSIZE))
#define  SELFP(b)       ((TREE **)(((uintptr_t)(b)) + SIZE(b)))
#define  LAST(b)        (*((TREE **)(((uintptr_t)(b)) - WORDSIZE)))
#define  NEXT(b)        ((TREE *)(((uintptr_t)(b)) + SIZE(b) + WORDSIZE))
```

Занятый блок:



Свободный блок:

Два младших бита SIZE кодируют занятость этого и предыдущего блока

Что если дескриптор не влезет в блок?

```
/* small blocks */
if (size < MINSIZE)
    return (_smalloc(size));
```

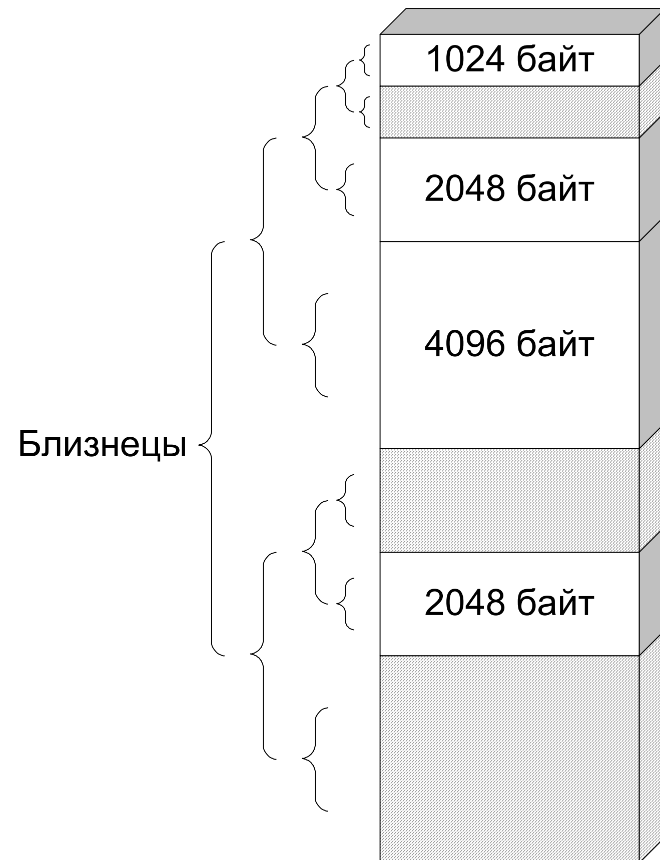
Маленькие блоки организованы в списки по размерам.
Если соответствующий список пуст, выделяется блок MINSIZE и режется на блоки нужного размера.
Маленькие блоки не имеют парной метки
и не могут быть слиты с большими или друг с другом.

Алгоритм близнецов (постановка задачи)

- First Fit не применим в задачах реального времени (в худшем случае требуется просмотр всего пула, ограничений на размер пула нет)
- (Дополнительно) Worst или Best Fit в сочетании с сортировкой пула по размеру
- Сортировка пула и поиск в сортированном пуле также зависят от размера пула – идея не подходит
- Вывод – необходимо ограничение на размер блока

Алгоритм близнецов

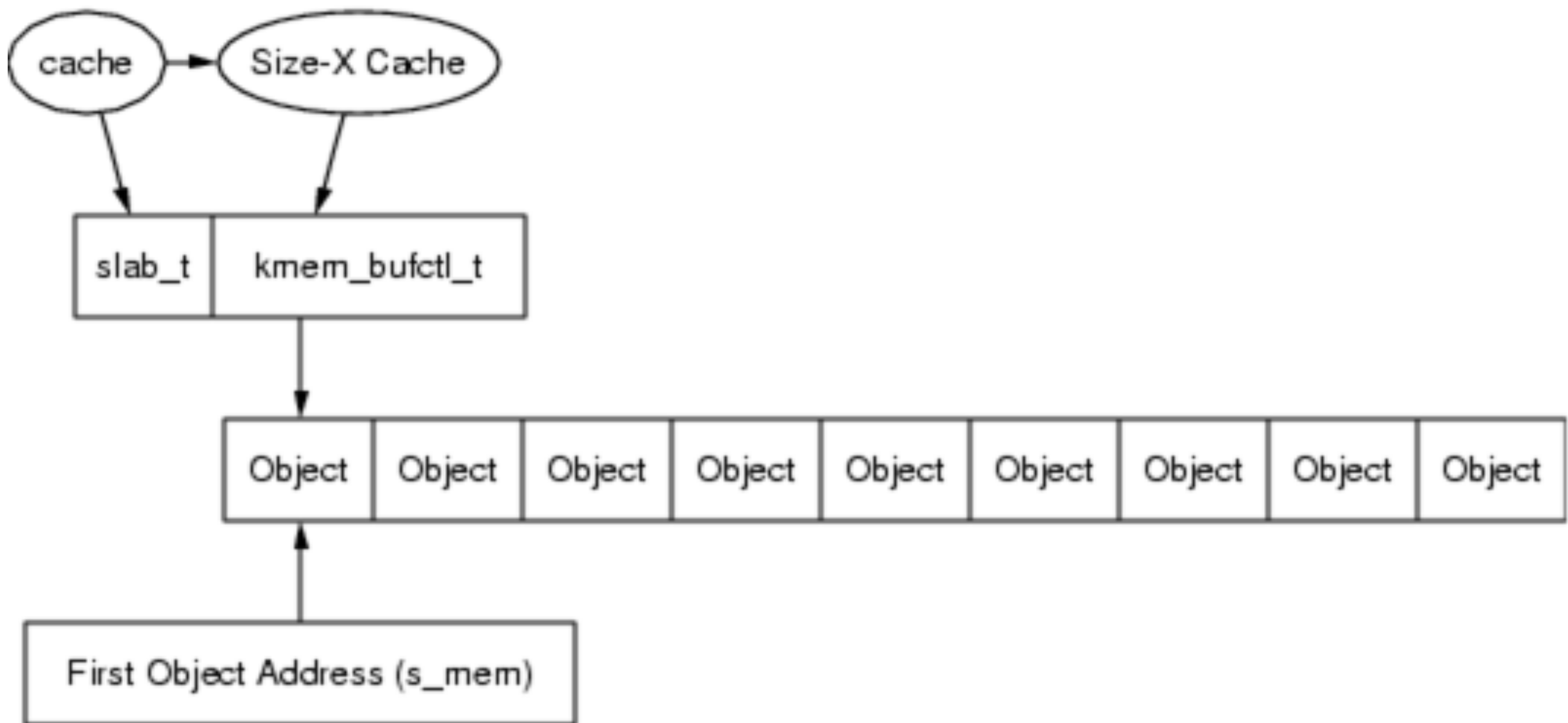
- Списки блоков фиксированных размеров
- Размеры пропорциональны степеням 2



Слабовые аллокаторы

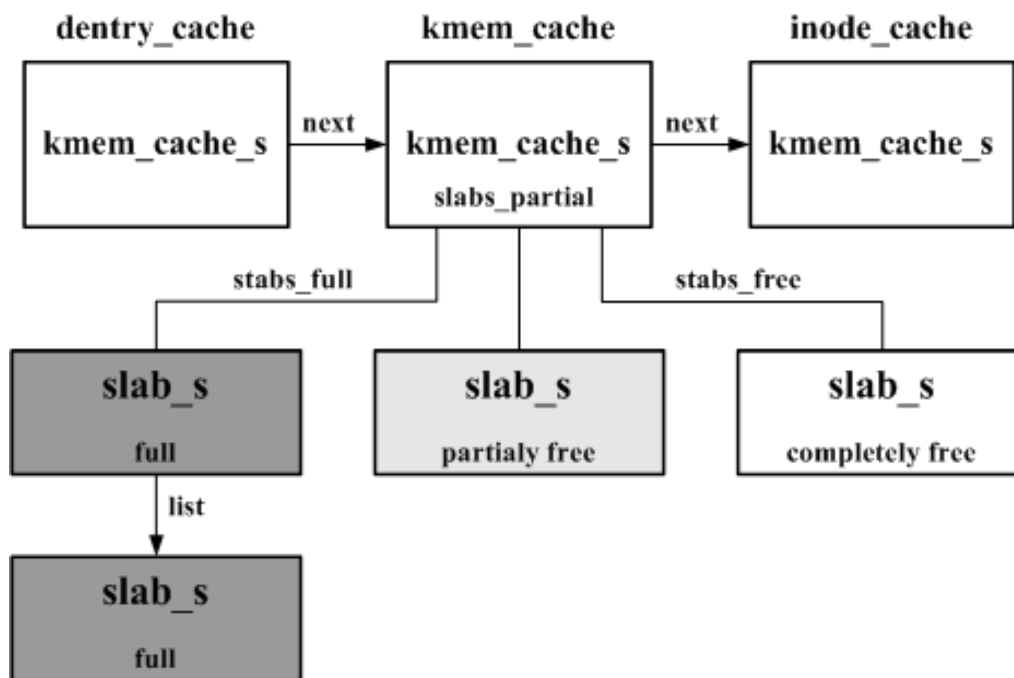
- Slab — плита, лист, пластина
- Впервые появились в ядре Solaris 2.4
- Используются, если вам нужно выделять много блоков одинакового размера
  - Особенно если вам может потребоваться реальное время и размер блоков не кратен 2^N
- Основная идея: получить большой кусок памяти и порезать его на много кусков требуемого размера.

Слаб



Слабовый аллокатор

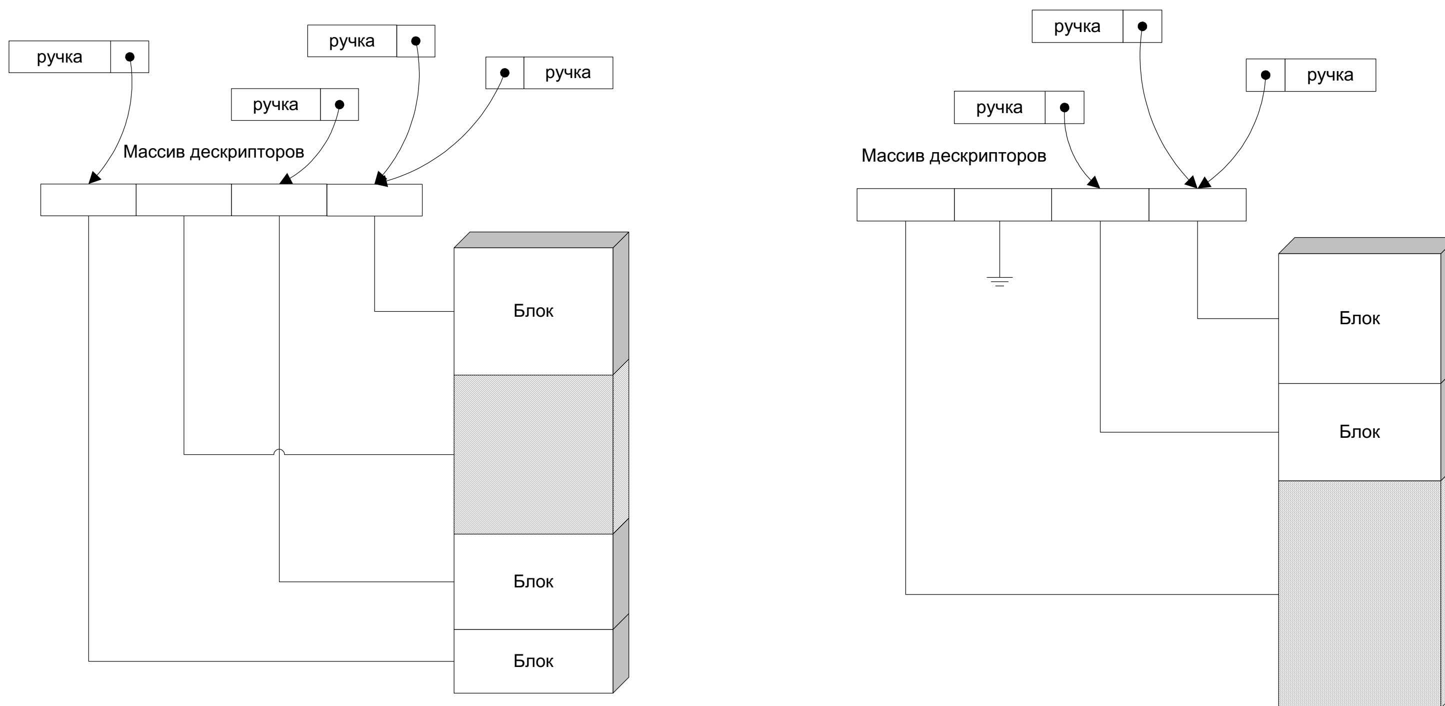
- Список слабов, порезанных на объекты одинакового размера, называется *кэшем*.



Дополнительно – «ручки»

- Попытка разрешить перемещение блоков по памяти
- При выделении памяти дается «ручка» (handle)
- Перед обращением к памяти ее надо заблокировать (GlobalLock – возвращает указатель)
- После завершения операции память надо разблокировать (GlobalUnlock). После этого система считает себя вправе перемещать объект по памяти
- Программный аналог аппаратных диспетчеров памяти. Применялись в Win16, MacOS до версии X

Дополнительно – дефрагментация при использовании «ручек»



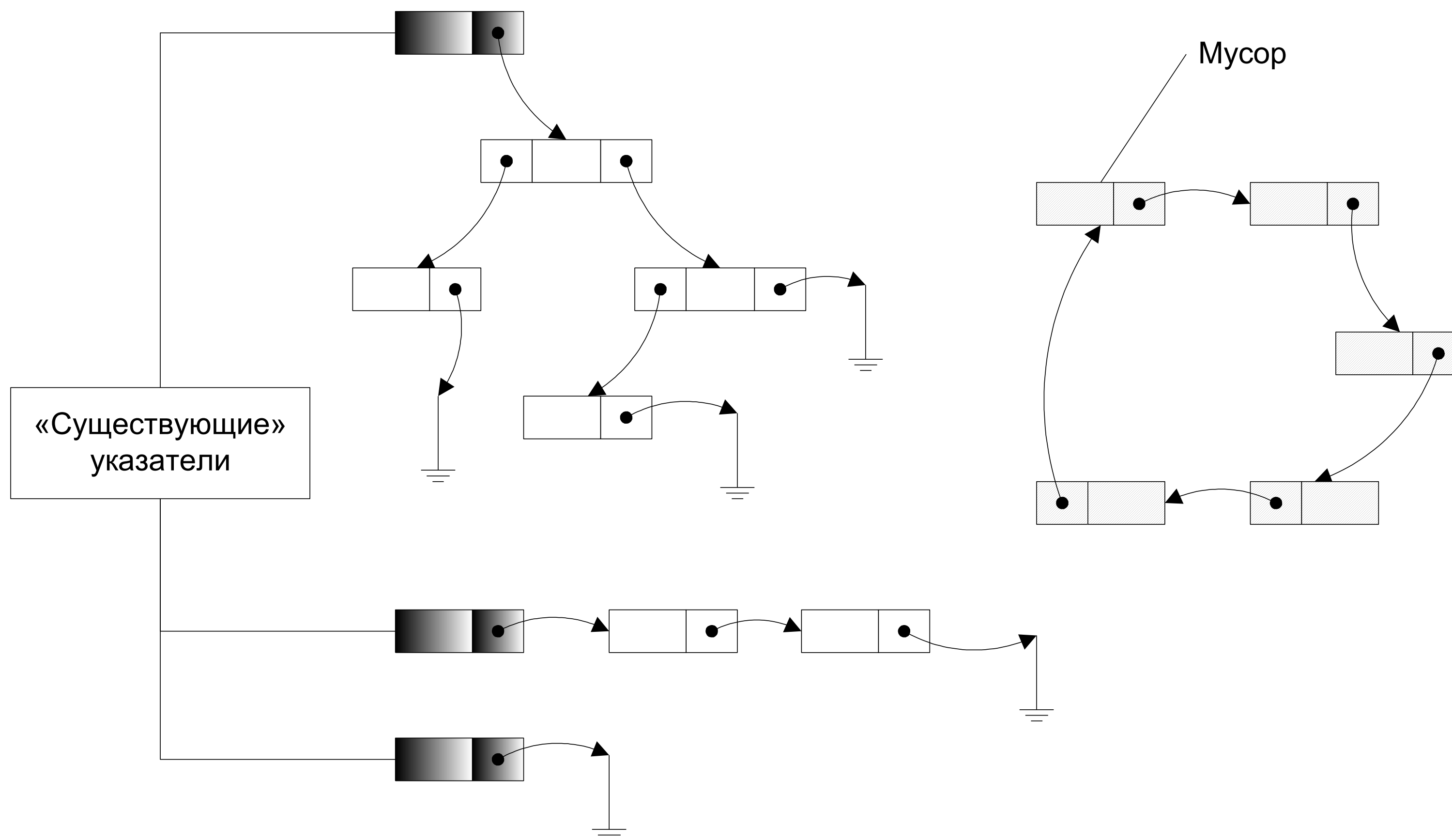
Дополнительно – «ручки» (продолжение)

- Недостаток – очень опасная ошибка, сохранение указателя после GlobalUnlock
  - Зависит от порядка обращений к памяти не только самой программы, но и других программ в системе
  - Как правило не может быть обнаружена при тестировании
  - «Система, которая работает иногда»
- Резкий рост стабильности той же смеси приложений после перехода от Win3.x к Win95
  - Предположение – в Win16 приложениях таких ошибок было очень много.

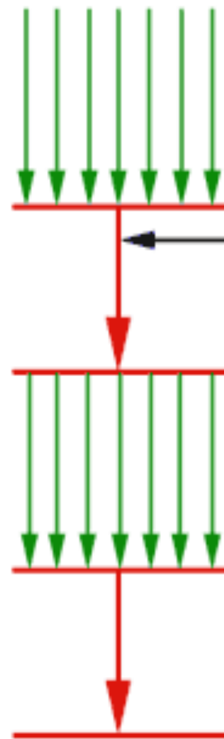
Неявное освобождение памяти (сборка мусора)

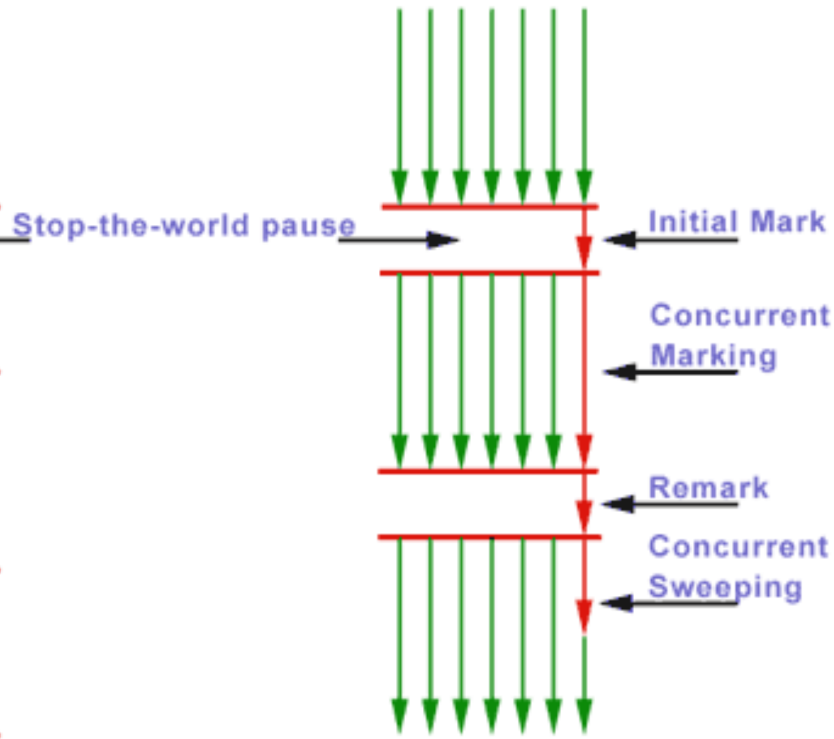
- Понятие утечки памяти
- Сборка мусора подсчетом ссылок (уходящий последним гасит свет)
  - Невозможна обработка циклических списков
  - Примеры – COM, Lotus Software Extension (LSX)
- Сборка мусора рекурсивным просмотром ссылок
  - В момент сборки необходимо останавливать всю деятельность, которая может привести к уничтожению и образованию ссылок и, тем более, объектов
  - Примеры – Java, Emacs Lisp
- Дополнительно – сравнение стоимости разработки (в человеко-часах) на C++ и Basic/Java

Сборка мусора просмотром ссылок

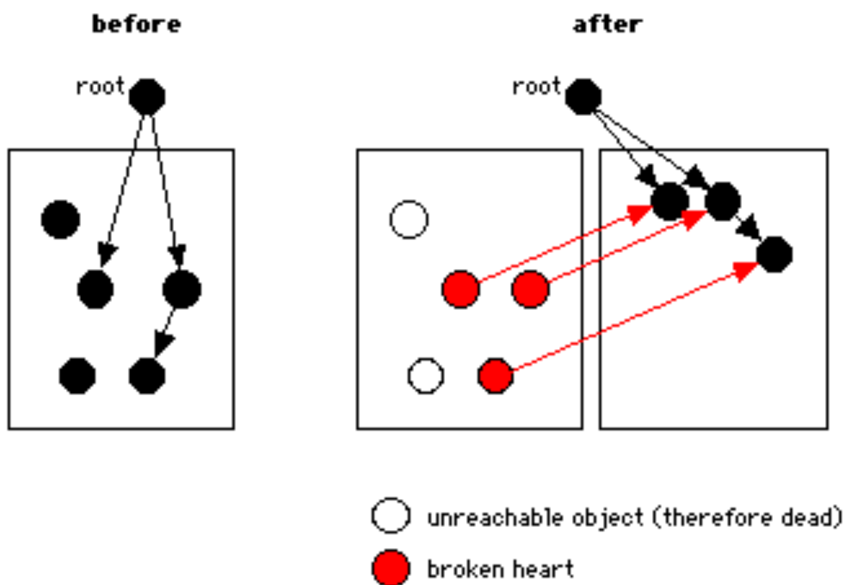


«Неблокирующийся» сборщик мусора

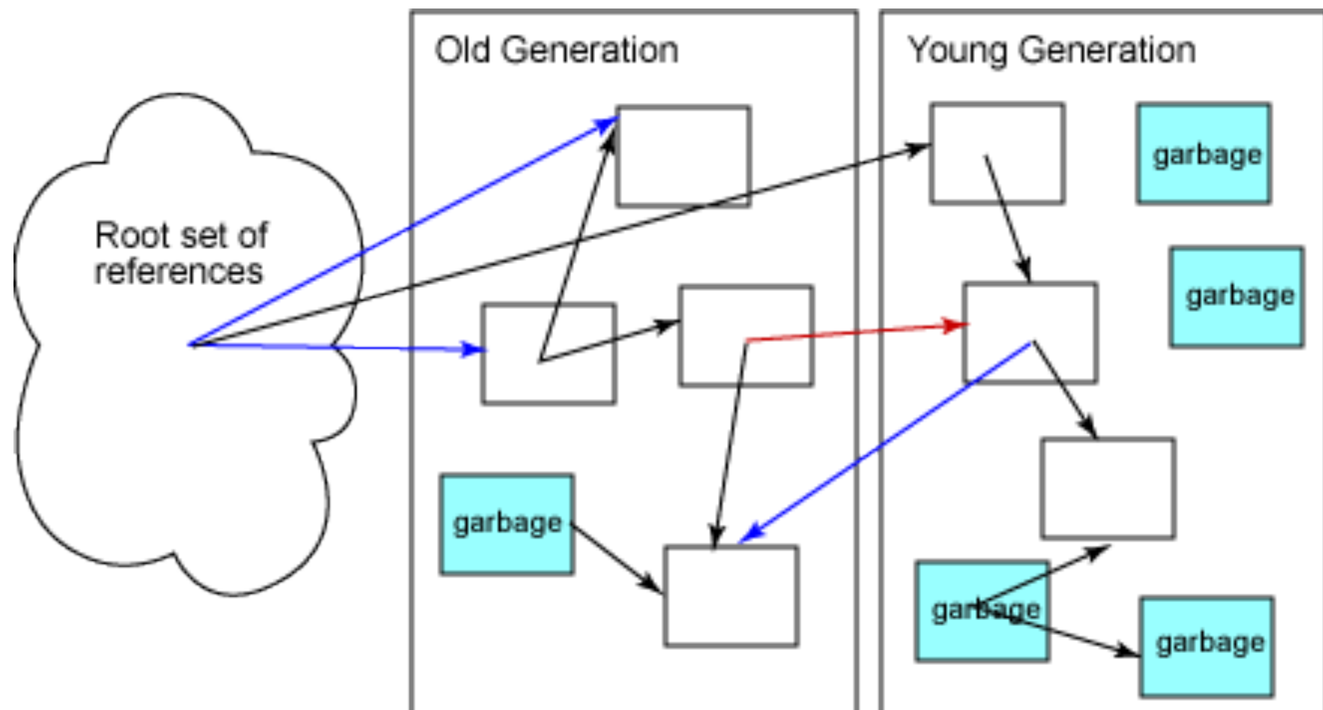




Копирующие сборщики мусора

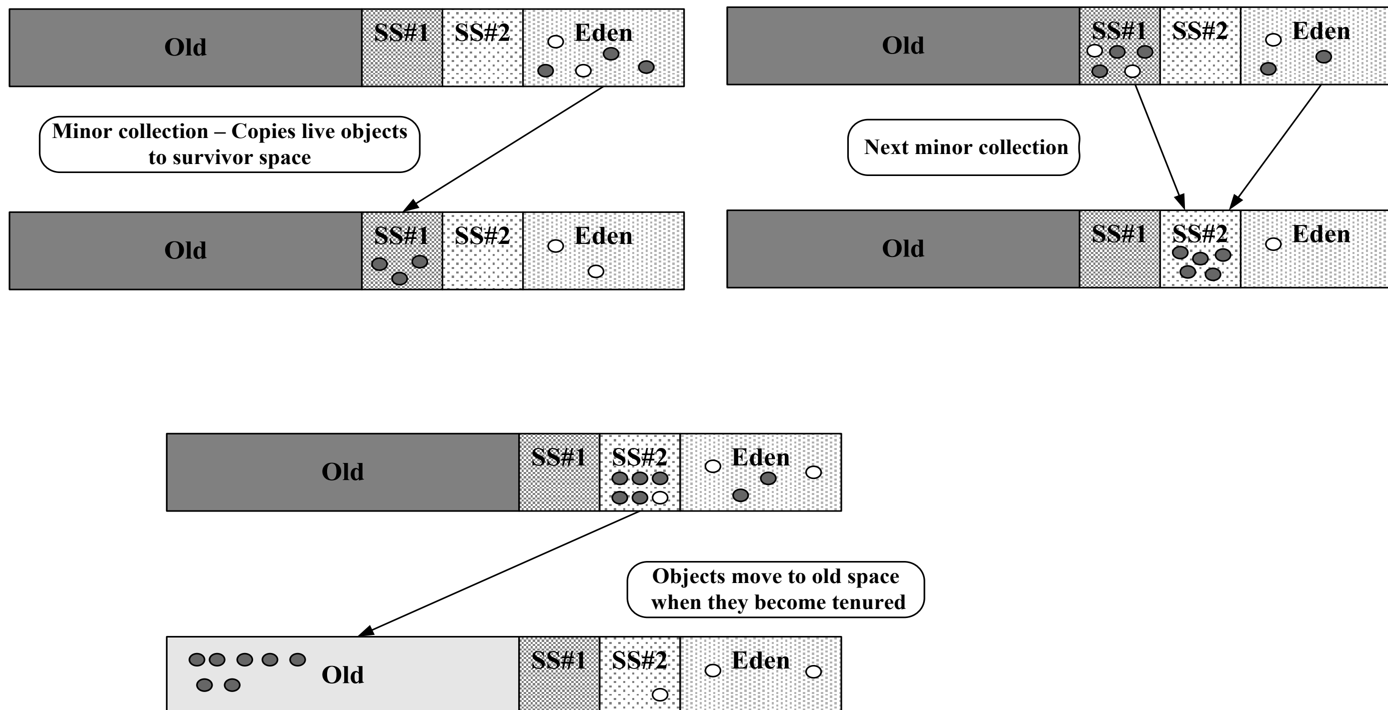


Взаимоотношения между долго- и короткоживущими объектами



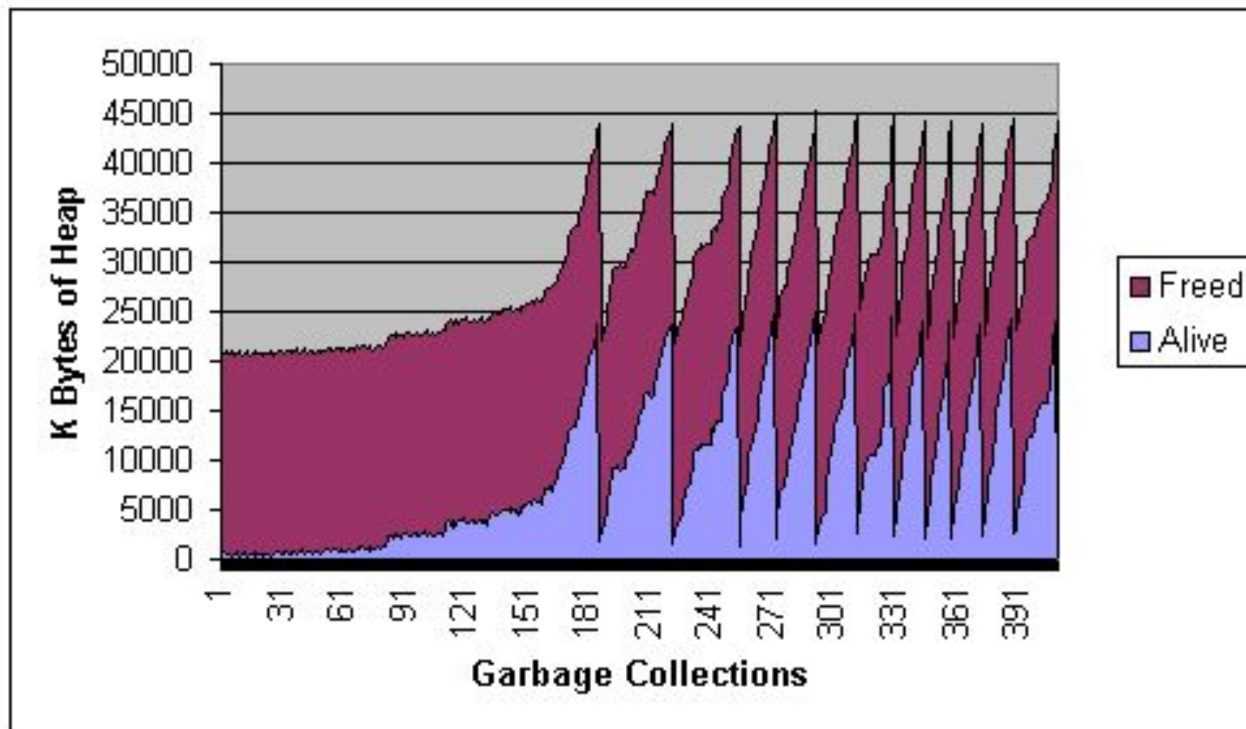
Генерационный сборщик мусора

http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-hotspotgc.html



Графики использования памяти (продолжение)

Изменения объема занятой памяти



Remembered set

- Генерационный сборщик основан на предположении, что «старые» объекты редко ссылаются на новые
- Поэтому при малых сборках «старые» объекты вообще не трогают.
- Что если «старый» объект все-таки получит ссылку на «новый»?
- Remembered set – это набор ссылок из «старых» объектов на «новые»

Garbage First (G1) коллектор

- Появился в Java 1.6
- Вместо «поколений» использует разбиение кучи на области («карты») одинакового размера
- Использует remembered set для отслеживания ссылок между картами
- Оценивает количество живых объектов в каждой карте

Garbage First (G1) коллектор

- Выбирает регионы с наименьшей стоимостью сборки (стоимость сборки оценивается через количество живых объектов и размер remembered set)
- Использует «неблокирующуюся» сборку

G1 - преимущества

- Выполняет большую часть работы в фоновом режиме
- Оптимизирован для работы на многоядерных процессорах
- Минимизирует паузы при сборке

G1 – недостатки

- Требует большого запаса памяти (как и все фоновые сборщики)
- Занимает процессор, что может быть существенно при малом числе процессорных ядер
- До некоторых объектов сборка может не доходить очень долго

RAII

Resource Allocation Is Initialization

```
{
    Window dialog(Window::modal, …);
    dialog.setText("bla-bla-bla");
     dialog.draw();
    ….
    dialog.~Window();
}
Window::~Window() {
    close(); free(…);
}
```

RAII и сборка мусора

- Фокус не проходит
- Финальный метод зовется не при выходе из блока, а неизвестно когда
- Необходим явный метод close() и явный его вызов.
- Висячие ссылки и утечки возвращаются!